

Labels: outcome (what you want to predict) 🡪 y

A **label** is the thing we're predicting—the y variable in simple linear regression. The label could be the future price of wheat, the kind of animal shown in a picture, the meaning of an audio clip, or just about anything.

Features: what are available inputs(x1, x2 ……., xn)

A **feature** is an input variable—the x variable in simple linear regression. A simple machine learning project might use a single feature, while a more sophisticated machine learning project could use millions of features, specified as:

x1,x2,...xN

Labeled data: both features and labels

Unlabeled data: only features without labels

Numerical data – we can use directly (float 32 or float 64)

Categorical data: change to numerical using binary or one hot encoding.

In the spam detector example, the features could include the following:

* words in the email text
* sender's address
* time of day the email was sent
* Email contains the phrase "one weird trick."

Model: Maps example to predict y

A model defines the relationship between features and label. For example, a spam detection model might associate certain features strongly with "spam". Let's highlight two phases of a model's life:

* **Training** means creating or **learning** the model. That is, you show the model labeled examples and enable the model to gradually learn the relationships between features and label.
* **Inference** means applying the trained model to unlabeled examples. That is, you use the trained model to make useful predictions (y'). For example, during inference, you can predict medianHouseValue for new unlabeled examples.

# Regression vs. classification

A **regression** model predicts continuous values. For example, regression models make predictions that answer questions like the following:

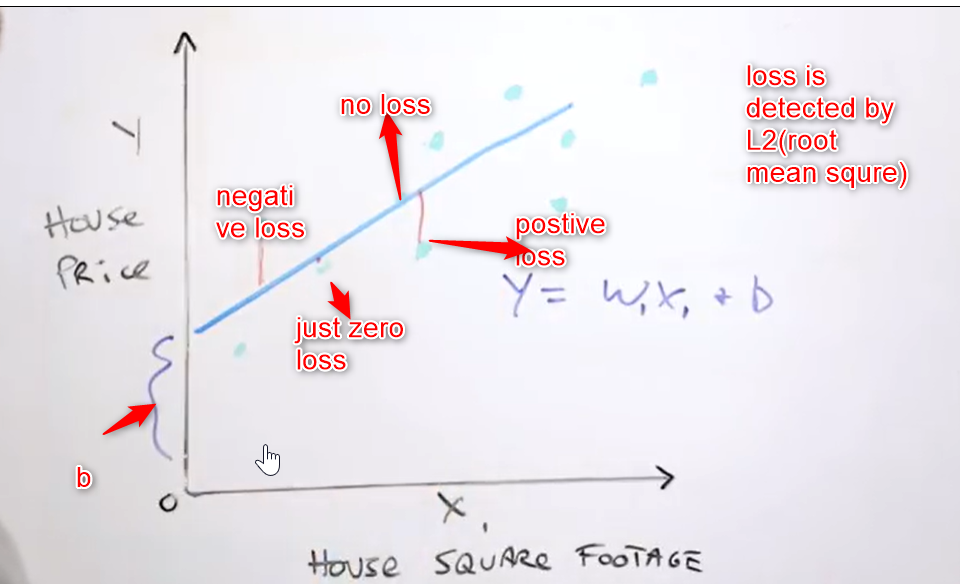
* What is the value of a house in California?
* What is the probability that a user will click on this ad?

A **classification** model predicts discrete values. For example, classification models make predictions that answer questions like the following:

* Is a given email message spam or not spam?
* Is this an image of a dog, a cat, or a hamster?

# Linear Regression:

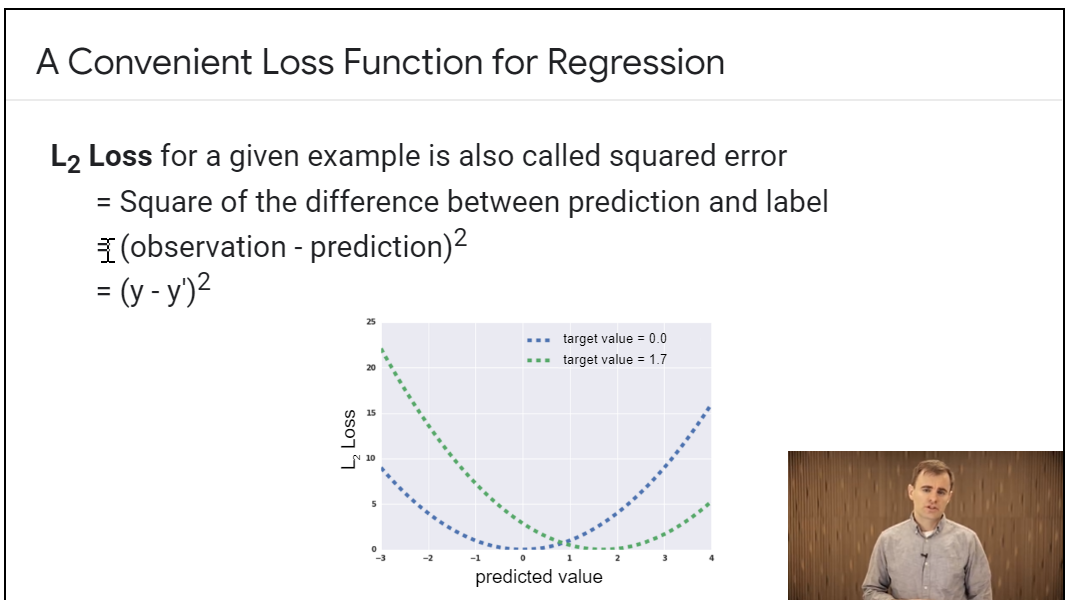
Continuous value

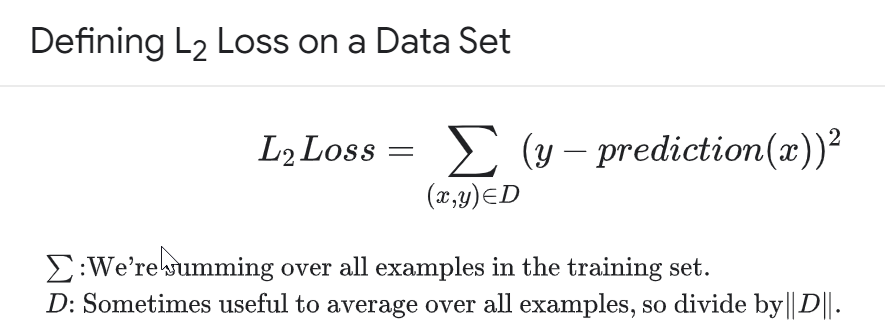


In W1X1 1 is dimension, b is bias

One of the loss function:

# L2 Loss:





# Eg:

X = chirps and y = temperature:

y=mx+b

where:

* y is the temperature in Celsius—the value we're trying to predict.
* m is the slope of the line.
* x is the number of chirps per minute—the value of our input feature.
* b is the y-intercept.

By convention in machine learning, you'll write the equation for a model slightly differently:

y′=b+w1x1

where:

* y′ is the predicted [label](https://developers.google.com/machine-learning/crash-course/framing/ml-terminology#labels) (a desired output).
* b is the bias (the y-intercept), sometimes referred to as w0.
* w1 is the weight of feature 1. Weight is the same concept as the "slope" m in the traditional equation of a line.
* x1 is a [feature](https://developers.google.com/machine-learning/crash-course/framing/ml-terminology#features) (a known input).

To **infer** (predict) the temperature y′ for a new chirps-per-minute value x1, just substitute the x1 value into this model.

Although this model uses only one feature, a more sophisticated model might rely on multiple features, each having a separate weight (w1, w2, etc.). For example, a model that relies on three features might look as follows:

y′=b+w1x1+w2x2+w3x3

# Computing Loss:

The linear regression models we'll examine here use a loss function called **squared loss** (also known as **L2 loss**). The squared loss for a single example is as follows:

= the square of the difference between the label and the prediction

= (observation - prediction(**x**))2

= (y - y')2

**Mean square error** (**MSE**) is the average squared loss per example over the whole dataset. To calculate MSE, sum up all the squared losses for individual examples and then divide by the number of examples:

MSE=(1\N)∑(x,y)∈D(y−prediction(x))2

where:

* (x,y) is an example in which
  + x is the set of features (for example, chirps/minute, age, gender) that the model uses to make predictions.
  + y is the example's label (for example, temperature).
* prediction(x) is a function of the weights and bias in combination with the set of features x.
* D is a data set containing many labeled examples, which are (x,y) pairs.
* N is the number of examples in D.

Although MSE is commonly-used in machine learning, it is neither the only practical loss function nor the best loss function for all circumstances.

# Use of loss function:

**Loss function**. In mathematical optimization and decision theory, a **loss function** or cost **function** is a **function** that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. An optimization problem seeks to minimize a **loss function**.

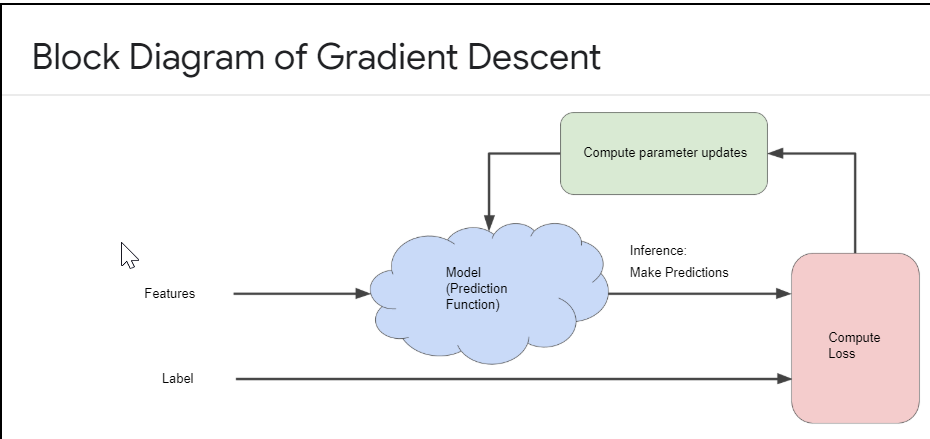
# Reducing or optimizing Loss:

Using Gradient decent:

Most of the Cases try to minimal (learning rate :) hyper parameter

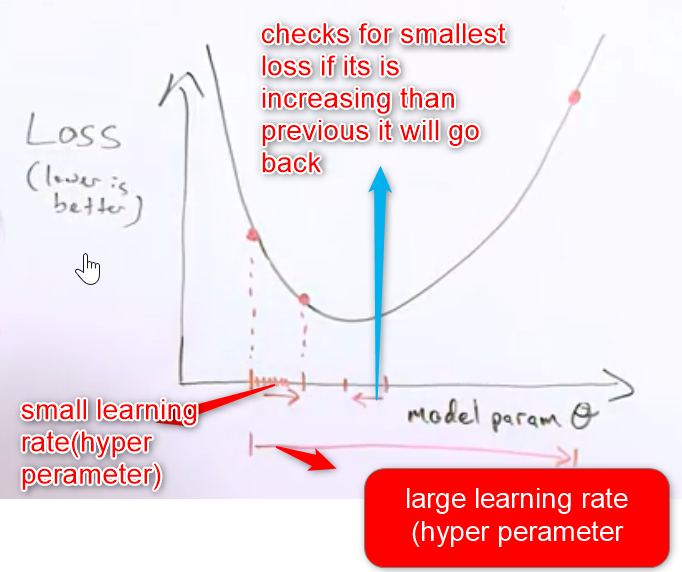
#check <https://developers.google.com/machine-learning/crash-course/fitter/graph>

**NOTE:** In practice, finding a "perfect" (or near-perfect) learning rate is not essential for successful model training. The goal is to find a learning rate large enough that gradient descent converges efficiently, but not so large that it never converges



## Stochastic gradient descent (SGD)

A [**gradient descent**](https://developers.google.com/machine-learning/glossary#gradient_descent) algorithm in which the batch size is one. In other words, SGD relies on a single example chosen uniformly at random from a dataset to calculate an estimate of the gradient at each step.



# **Generalization**

**Generalization** refers to your model's ability to adapt properly to new, previously unseen data, drawn from the same distribution as the one used to create the model.

* Develop intuition about overfitting.
* Determine whether a model is good or not.
* Divide a data set into a training set and a test set

Refers to your model's ability to make correct predictions on new, previously unseen data as opposed to the data used to train the model.

Use regularization to reduce overfitting

## Backpropagation

The primary algorithm for performing [**gradient descent**](https://developers.google.com/machine-learning/glossary#gradient_descent) on [**neural networks**](https://developers.google.com/machine-learning/glossary#neural_network). First, the output values of each node are calculated (and cached) in a forward pass. Then, the [**partial derivative**](https://developers.google.com/machine-learning/glossary#partial_derivative) of the error with respect to each parameter is calculated in a backward pass through the graph.

## Overfitting

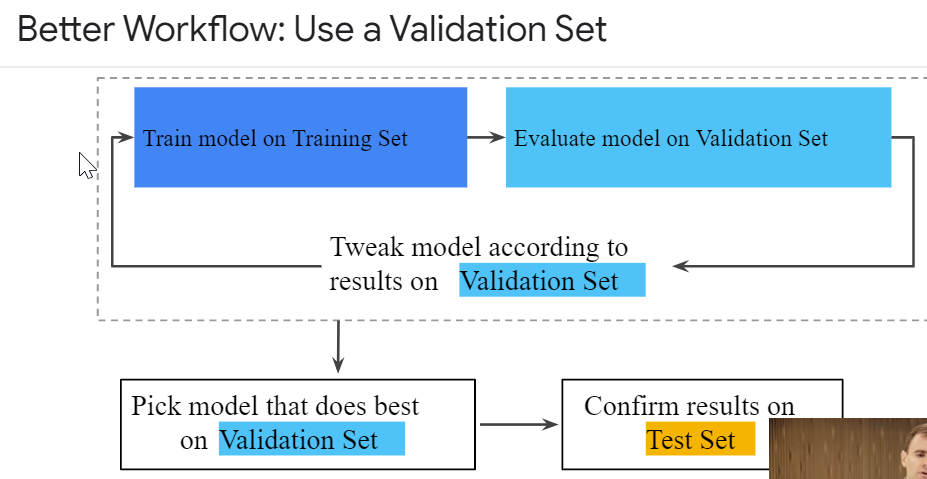
Creating a model that matches the [**training data**](https://developers.google.com/machine-learning/glossary#training_set) so closely that the model fails to make correct predictions on new data.

## Prediction

A model's output when provided with an input [**example**](https://developers.google.com/machine-learning/glossary#example).

# **Validation Set**

Partitioning a data set into a training set and test set lets you judge whether a given model will generalize well to new data. However, using only two partitions may be insufficient when doing many rounds of hyper parameter tuning.



# **Representation**

A machine learning model can't directly see, hear, or sense input examples. Instead, you must create a **representation** of the data to provide the model with a useful vantage point into the data's key qualities. That is, in order to train a model, you must choose the set of features that best represent the data.

## Feature engineering

The process of determining which [**features**](https://developers.google.com/machine-learning/glossary#feature) might be useful in training a model, and then converting raw data from log files and other sources into said features. In TensorFlow, feature engineering often means converting raw log file entries to [**tf.Example**](https://developers.google.com/machine-learning/glossary#tf.Example) protocol buffers. See also [tf.Transform](https://github.com/tensorflow/transform).

Feature engineering is sometimes called **feature extraction**.

## Feature extraction

Overloaded term having either of the following definitions:

* Retrieving intermediate feature representations calculated by an [**unsupervised**](https://developers.google.com/machine-learning/glossary#unsupervised_machine_learning) or pretrained model (for example, [**hidden layer**](https://developers.google.com/machine-learning/glossary#hidden_layer) values in a [**neural network**](https://developers.google.com/machine-learning/glossary#neural_network)) for use in another model as input.
* Synonym for [**feature engineering**](https://developers.google.com/machine-learning/glossary#feature_engineering).

## Feature set

The group of [**features**](https://developers.google.com/machine-learning/glossary#feature) your machine learning model trains on. For example, postal code, property size, and property condition might comprise a simple feature set for a model that predicts housing prices.

## Feature spec

#TensorFlow

Describes the information required to extract [**features**](https://developers.google.com/machine-learning/glossary#feature) data from the [**tf.Example**](https://developers.google.com/machine-learning/glossary#tf.Example) protocol buffer. Because the tf.Example protocol buffer is just a container for data, you must specify the following:

* the data to extract (that is, the keys for the features)
* the data type (for example, float or int)
* The length (fixed or variable)

The [**Estimator API**](https://developers.google.com/machine-learning/glossary#Estimators) provides facilities for producing a feature spec from a list of [**FeatureColumns**](https://developers.google.com/machine-learning/glossary#feature_columns).

## Feature vector

The list of feature values representing an [**example**](https://developers.google.com/machine-learning/glossary#example) passed into a model.

## Mapping Raw Data to Features

The left side of Figure 1 illustrates raw data from an input data source; the right side illustrates a **feature vector**, which is the set of floating-point values comprising the examples in your data set. **Feature engineering** means transforming raw data into a feature vector. Expect to spend significant time doing feature engineering.

Many machine learning models must represent the features as real-numbered vectors since the feature values must be multiplied by the model weights.

### **Mapping numeric values**

Integer and floating-point data don't need a special encoding because they can be multiplied by a numeric weight. As suggested in Figure 2, converting the raw integer value 6 to the feature value 6.0 is trivial:

### **Mapping categorical values**

[Categorical features](https://developers.google.com/machine-learning/glossary#categorical_data) have a discrete set of possible values. For example, there might be a feature called street\_name with options that include:

{'Charleston Road', 'North Shoreline Boulevard', 'Shorebird Way', 'Rengstorff Avenue'}

The length of this vector is equal to the number of elements in the vocabulary. This representation is called a **one-hot encoding** when a single value is 1, and a **multi-hot encoding** when multiple values are 1.

## One-hot encoding

A sparse vector in which:

* One element is set to 1.
* All other elements are set to 0.

One-hot encoding is commonly used to represent strings or identifiers that have a finite set of possible values. For example, suppose a given botany dataset chronicles 15,000 different species, each denoted with a unique string identifier. As part of feature engineering, you'll probably encode those string identifiers as one-hot vectors in which the vector has a size of 15,000.

## Outliers

Values distant from most other values. In machine learning, any of the following are outliers (Extreme Values)

* [**Weights**](https://developers.google.com/machine-learning/glossary#weight) with high absolute values.
* Predicted values relatively far away from the actual values.
* Input data whose values are more than roughly 3 standard deviations from the mean.

Outliers often cause problems in model training. [**Clipping**](https://developers.google.com/machine-learning/glossary#clipping) is one way of managing outliers.

# **Feature Crosses**

A **feature cross** is a **synthetic feature** formed by multiplying (crossing) two or more features. Crossing combinations of features can provide predictive abilities beyond what those features can provide individually.

X1, x2 are features use x1^2, x2^2 and X1X2

To get Non linearity use feature cross product.

# **Synthetic** **feature**

A [**feature**](https://developers.google.com/machine-learning/glossary#feature) not present among the input features, but created from one or more of them. Kinds of synthetic features include:

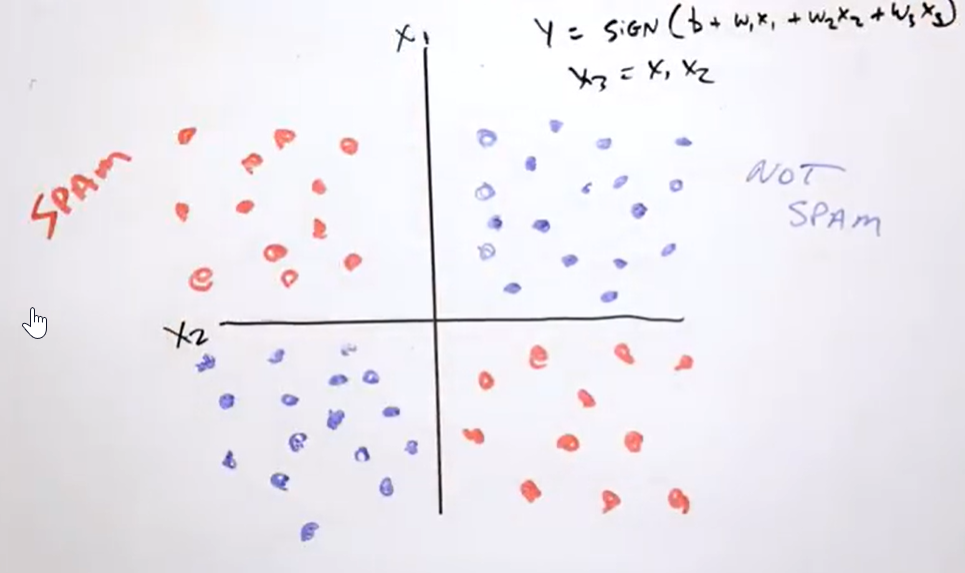
* [**Bucketing**](https://developers.google.com/machine-learning/glossary#bucketing) a continuous feature into range bins.
* Multiplying (or dividing) one feature value by other feature value(s) or by itself.
* Creating a [**feature cross**](https://developers.google.com/machine-learning/glossary#feature_cross).

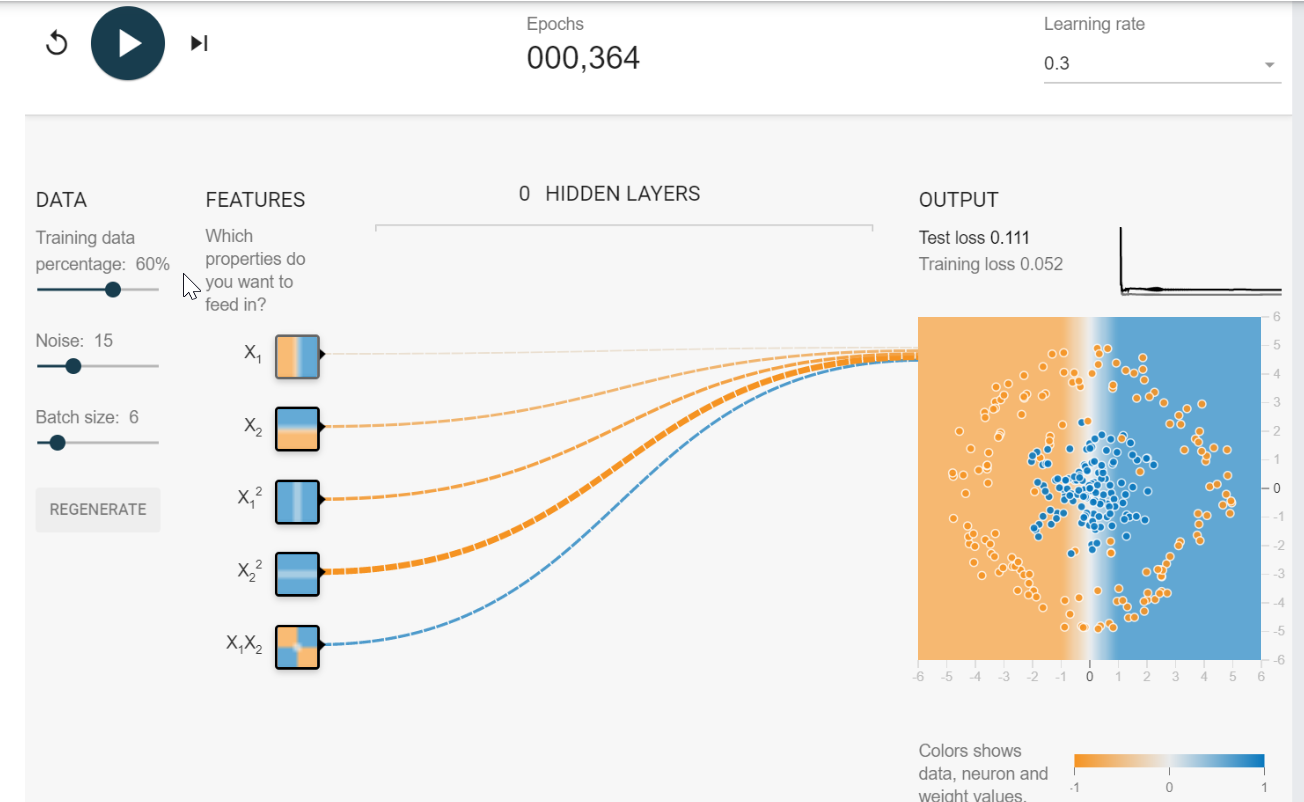
Features created by [**normalizing**](https://developers.google.com/machine-learning/glossary#normalization) or [**scaling**](https://developers.google.com/machine-learning/glossary#scaling) alone are not considered synthetic features.

If we have the data set like given below even though its linear problem it is difficult to train so we will use x3 as the product of (x1 and x2) called feature crosses to solve these kinds of problems

Non linearity in linear model is called synthetic feature = x3

Note: add x1 square and x2 square to get the best results





## Feature Crosses

Crossing two (or more) features is a clever way to learn non-linear relations using a linear model. In our problem, if we just use the feature latitude for learning, the model might learn that city blocks at a particular latitude (or within a particular range of latitudes since we have bucketized it) are more likely to be expensive than others. Similarly for the feature longitude. However, if we cross longitude by latitude, the crossed feature represents a well defined city block. If the model learns that certain city blocks (within range of latitudes and longitudes) are more likely to be more expensive than others, it is a stronger signal than two features considered individually.

# **Regularization for Simplicity: L₂ Regularization**

Consider the following **generalization curve**, which shows the loss for both the training set and validation set against the number of training iterations.



**Figure 1. Loss on training set and validation set.**

Figure 1 shows a model in which training loss gradually decreases, but validation loss eventually goes up. In other words, this generalization curve shows that the model is [overfitting](https://developers.google.com/machine-learning/crash-course/generalization/peril-of-overfitting) to the data in the training set. Channeling our inner [Ockham](https://developers.google.com/machine-learning/crash-course/generalization/peril-of-overfitting#ockham), perhaps we could prevent overfitting by penalizing complex models, a principle called **regularization**.

In other words, instead of simply aiming to minimize loss (empirical risk minimization):

minimize(Loss(Data|Model))

we'll now minimize loss+complexity, which is called **structural risk minimization**:

minimize(Loss(Data|Model) + complexity(Model))

Our training optimization algorithm is now a function of two terms: the **loss term**, which measures how well the model fits the data, and the **regularization term**, which measures model complexity.

Machine Learning Crash Course focuses on two common (and somewhat related) ways to think of model complexity:

* Model complexity as a function of the weights of all the features in the model.
* Model complexity as a function of the total number of features with nonzero weights. (A [later module](https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/l1-regularization) covers this approach.)

If model complexity is a function of weights, a feature weight with a high absolute value is more complex than a feature weight with a low absolute value.

We can quantify complexity using the **L2 regularization** formula, which defines the regularization term as the sum of the squares of all the feature weights:

L2 regularization term=||w||22=w12+w22+...+wn2

In this formula, weights close to zero have little effect on model complexity, while outlier weights can have a huge impact.

For example, a linear model with the following weights:

{w1=0.2,w2=0.5,w3=5,w4=1,w5=0.25,w6=0.75}

Has an *L2* regularization term of 26.915:

w12+w22+w32+w42+w52+w62

=0.22+0.52+52+12+0.252+0.752

=0.04+0.25+25+1+0.0625+0.5625

=26.915

But w3 (bolded above), with a squared value of 25, contributes nearly all the complexity. The sum of the squares of all five other weights adds just 1.915 to the *L2* regularization term.

## Generalization curve

A [**loss curve**](https://developers.google.com/machine-learning/glossary#loss_curve) showing both the [**training set**](https://developers.google.com/machine-learning/glossary#training_set) and the [**validation set**](https://developers.google.com/machine-learning/glossary#validation_set). A generalization curve can help you detect possible [**overfitting**](https://developers.google.com/machine-learning/glossary#overfitting). For example, the following generalization curve suggests overfitting because loss for the validation set ultimately becomes significantly higher than for the training set.



# L1 loss

[**Loss**](https://developers.google.com/machine-learning/glossary#loss) function based on the absolute value of the difference between the values that a model is predicting and the actual values of the [**labels**](https://developers.google.com/machine-learning/glossary#label). L1 loss is less sensitive to outliers than [**L2 loss**](https://developers.google.com/machine-learning/glossary#squared_loss).

# L1 regularization

A type of [**regularization**](https://developers.google.com/machine-learning/glossary#regularization) that penalizes weights in proportion to the sum of the absolute values of the weights. In models relying on [**sparse features**](https://developers.google.com/machine-learning/glossary#sparse_features), L1 regularization helps drive the weights of irrelevant or barely relevant features to exactly 0, which removes those features from the model. Contrast with [**L2 regularization**](https://developers.google.com/machine-learning/glossary#L2_regularization).

# L2 loss

See [**squared loss**](https://developers.google.com/machine-learning/glossary#squared_loss).

# L2 regularization

A type of [**regularization**](https://developers.google.com/machine-learning/glossary#regularization) that penalizes weights in proportion to the sum of the squares of the weights. L2 regularization helps drive outlier weights (those with high positive or low negative values) closer to 0 but not quite to 0. (Contrast with [**L1 regularization**](https://developers.google.com/machine-learning/glossary#L1_regularization).) L2 regularization always improves generalization in linear models.

# Regularization

The penalty on a model's complexity. Regularization helps prevent [**overfitting**](https://developers.google.com/machine-learning/glossary#overfitting). Different kinds of regularization include:

* [**L1 regularization**](https://developers.google.com/machine-learning/glossary#L1_regularization)
* [**L2 regularization**](https://developers.google.com/machine-learning/glossary#L2_regularization)
* [**dropout regularization**](https://developers.google.com/machine-learning/glossary#dropout_regularization)
* [**early stopping**](https://developers.google.com/machine-learning/glossary#early_stopping) (this is not a formal regularization method, but can effectively limit overfitting)

# Regularization rate

A scalar value, represented as lambda, specifying the relative importance of the regularization function. The following simplified [**loss**](https://developers.google.com/machine-learning/glossary#loss) equation shows the regularization rate's influence:

minimize(loss function + λ(regularization function))

Raising the regularization rate reduces [**overfitting**](https://developers.google.com/machine-learning/glossary#overfitting) but may make the model less [**accurate**](https://developers.google.com/machine-learning/glossary#accuracy).

# **Regularization for Simplicity: Lambda**

**Estimated Time:** 8 minutes

Model developers tune the overall impact of the regularization term by multiplying its value by a scalar known as **lambda** (also called the **regularization rate**). That is, model developers aim to do the following:

minimize(Loss(Data|Model)+λ complexity(Model))

Performing *L2* regularization has the following effect on a model

* Encourages weight values toward 0 (but not exactly 0)
* Encourages the mean of the weights toward 0, with a normal (bell-shaped or Gaussian) distribution.

Increasing the lambda value strengthens the regularization effect. For example, the histogram of weights for a high value of lambda might look as shown in Figure 2.

**Figure 2. Histogram of weights.**

Lowering the value of lambda tends to yield a flatter histogram, as shown in Figure 3.

**Figure 3. Histogram of weights produced by a lower lambda value.**

When choosing a lambda value, the goal is to strike the right balance between simplicity and training-data fit:

* If your lambda value is too high, your model will be simple, but you run the risk of underfitting your data. Your model won't learn enough about the training data to make useful predictions.
* If your lambda value is too low, your model will be more complex, and you run the risk of overfitting your data. Your model will learn too much about the particularities of the training data, and won't be able to generalize to new data.

**Note:** Setting lambda to zero removes regularization completely. In this case, training focuses exclusively on minimizing loss, which poses the highest possible overfitting risk.

The ideal value of lambda produces a model that generalizes well to new, previously unseen data. Unfortunately, that ideal value of lambda is data-dependent, so you'll need to do some tuning.

# **Lambda**

Synonym for [**regularization rate**](https://developers.google.com/machine-learning/glossary#regularization_rate).

# **Logistic Regression**

Instead of predicting exactly 0 or 1, **logistic regression** generates a probability—a value between 0 and 1, exclusive. For example, consider a logistic regression model for spam detection. If the model infers a value of 0.932 on a particular email message, it implies a 93.2% probability that the email message is spam. More precisely, it means that in the limit of infinite training examples, the set of examples for which the model predicts 0.932 will actually be spam 93.2% of the time and the remaining 6.8% will not.

## Binary classification

A type of [**classification**](https://developers.google.com/machine-learning/glossary#classification_model) task that outputs one of two mutually exclusive [**classes**](https://developers.google.com/machine-learning/glossary#class). For example, a machine learning model that evaluates email messages and outputs either "spam" or "not spam" is a [**binary classifier**](https://developers.google.com/machine-learning/glossary#binary_classification).

## Loss function for Logistic Regression

The loss function for linear regression is squared loss. The loss function for logistic regression is **Log Loss**, which is defined as follows:

Log Loss=∑(x,y)∈D−ylog⁡(y′)−(1−y)log⁡(1−y′)

where:

* (x,y)∈D is the data set containing many labeled examples, which are (x,y) pairs.
* y is the label in a labeled example. Since this is logistic regression, every value of y must either be 0 or 1.
* y′ is the predicted value (somewhere between 0 and 1), given the set of features in x.

## Regularization in Logistic Regression

[Regularization](https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/video-lecture) is extremely important in logistic regression modeling. Without regularization, the asymptotic nature of logistic regression would keep driving loss towards 0 in high dimensions. Consequently, most logistic regression models use one of the following two strategies to dampen model complexity:

* L2 regularization.
* Early stopping, that is, limiting the number of training steps or the learning rate.

(We'll discuss a third strategy—L1 regularization—in a [later module](https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/video-lecture).)

Imagine that you assign a unique id to each example, and map each id to its own feature. If you don't specify a regularization function, the model will become completely overfit. That's because the model would try to drive loss to zero on all examples and never get there, driving the weights for each indicator feature to +infinity or -infinity. This can happen in high dimensional data with feature crosses, when there’s a huge mass of rare crosses that happen only on one example each.

Fortunately, using L2 or early stopping will prevent this problem.

## Bias (ethics/fairness)

#fairness

1. Stereotyping, prejudice or favoritism towards some things, people, or groups over others. These biases can affect collection and interpretation of data, the design of a system, and how users interact with a system. Forms of this type of bias include:

* [**automation bias**](https://developers.google.com/machine-learning/glossary#automation_bias)
* [**confirmation bias**](https://developers.google.com/machine-learning/glossary#confirmation_bias)
* [**experimenter’s bias**](https://developers.google.com/machine-learning/glossary#confirmation_bias)
* [**group attribution bias**](https://developers.google.com/machine-learning/glossary#group_attribution_bias)
* [**implicit bias**](https://developers.google.com/machine-learning/glossary#implicit_bias)
* [**in-group bias**](https://developers.google.com/machine-learning/glossary#in-group_bias)
* [**out-group homogeneity bias**](https://developers.google.com/machine-learning/glossary#out-group_homogeneity_bias)

2. Systematic error introduced by a sampling or reporting procedure. Forms of this type of bias include:

* [**coverage bias**](https://developers.google.com/machine-learning/glossary#selection_bias)
* [**non-response bias**](https://developers.google.com/machine-learning/glossary#selection_bias)
* [**participation bias**](https://developers.google.com/machine-learning/glossary#participation_bias)
* [**reporting bias**](https://developers.google.com/machine-learning/glossary#reporting_bias)
* [**sampling bias**](https://developers.google.com/machine-learning/glossary#selection_bias)
* [**selection bias**](https://developers.google.com/machine-learning/glossary#selection_bias)

Not to be confused with the [bias term](https://developers.google.com/machine-learning/glossary#bias) in machine learning models or [**prediction bias**](https://developers.google.com/machine-learning/glossary#prediction_bias).

## Bias (math)

An intercept or offset from an origin. Bias (also known as the **bias term**) is referred to as b or *w0* in machine learning models. For example, bias is the b in the following formula:

y′=b+w1x1+w2x2+…wnxn

Not to be confused with [**bias in ethics and fairness**](https://developers.google.com/machine-learning/glossary#bias_ethics) or [**prediction bias**](https://developers.google.com/machine-learning/glossary#prediction_bias).

## accuracy

The fraction of [**predictions**](https://developers.google.com/machine-learning/glossary#prediction) that a [**classification model**](https://developers.google.com/machine-learning/glossary#classification_model) got right. In [**multi-class classification**](https://developers.google.com/machine-learning/glossary#multi-class), accuracy is defined as follows:

Accuracy=Correct Predictions/Total Number Of Examples

In [**binary classification**](https://developers.google.com/machine-learning/glossary#binary_classification), accuracy has the following definition:

Accuracy=True Positives+True Negatives/Total Number Of Examples

See [**true positive**](https://developers.google.com/machine-learning/glossary#TP) and [**true negative**](https://developers.google.com/machine-learning/glossary#TN).

## Classification model

A type of machine learning model for distinguishing among two or more discrete classes. For example, a natural language processing classification model could determine whether an input sentence was in French, Spanish, or Italian. Compare with [**regression model**](https://developers.google.com/machine-learning/glossary#regression_model).

# **Classification: True vs. False and Positive vs. Negative**

**An Aesop's Fable: The Boy Who Cried Wolf (compressed)**

A shepherd boy gets bored tending the town's flock. To have some fun, he cries out, "Wolf!" even though no wolf is in sight. The villagers run to protect the flock, but then get really mad when they realize the boy was playing a joke on them.

[Iterate previous paragraph N times.]

One night, the shepherd boy sees a real wolf approaching the flock and calls out, "Wolf!" The villagers refuse to be fooled again and stay in their houses. The hungry wolf turns the flock into lamb chops. The town goes hungry. Panic ensues.

Let's make the following definitions:

* "Wolf" is a **positive class**.
* "No wolf" is a **negative class**.

We can summarize our "wolf-prediction" model using a 2x2 [confusion matrix](https://developers.google.com/machine-learning/glossary#confusion_matrix) that depicts all four possible outcomes:

|  |  |
| --- | --- |
| True Positive (TP):   * Reality: A wolf threatened. * Shepherd said: "Wolf." * Outcome: Shepherd is a hero. | False Positive (FP):   * Reality: No wolf threatened. * Shepherd said: "Wolf." * Outcome: Villagers are angry at shepherd for waking them up. |
| False Negative (FN):   * Reality: A wolf threatened. * Shepherd said: "No wolf." * Outcome: The wolf ate all the sheep. | True Negative (TN):   * Reality: No wolf threatened. * Shepherd said: "No wolf." * Outcome: Everyone is fine. |

A **true positive** is an outcome where the model correctly predicts the positive class. Similarly, a **true negative** is an outcome where the model correctly predicts the negative class.

A **false positive** is an outcome where the model incorrectly predicts the positive class. And a **false negative** is an outcome where the model incorrectly predicts the negative class.

## Confusion matrix

An NxN table that summarizes how successful a [**classification model's**](https://developers.google.com/machine-learning/glossary#classification_model) predictions were; that is, the correlation between the label and the model's classification. One axis of a confusion matrix is the [**label**](https://developers.google.com/machine-learning/glossary#label) that the model predicted, and the other axis is the actual label. N represents the number of [**classes**](https://developers.google.com/machine-learning/glossary#class). In a [**binary classification**](https://developers.google.com/machine-learning/glossary#binary_classification) problem, N=2. For example, here is a sample confusion matrix for a binary classification problem:

|  | Tumor (predicted) | Non-Tumor (predicted) |
| --- | --- | --- |
| Tumor (actual) | 18 | 1 |
| Non-Tumor (actual) | 6 | 452 |

The preceding confusion matrix shows that of the 19 samples that actually had tumors, the model correctly classified 18 as having tumors (18 [**true positives**](https://developers.google.com/machine-learning/glossary#TP)), and incorrectly classified 1 as not having a tumor (1 [**false negative**](https://developers.google.com/machine-learning/glossary#FN)). Similarly, of 458 samples that actually did not have tumors, 452 were correctly classified (452 [**true negatives**](https://developers.google.com/machine-learning/glossary#TN)) and 6 were incorrectly classified (6 [**false positives**](https://developers.google.com/machine-learning/glossary#FP)).

The confusion matrix for a [**multi-class classification**](https://developers.google.com/machine-learning/glossary#multi-class) problem can help you determine mistake patterns. For example, a confusion matrix could reveal that a model trained to recognize handwritten digits tends to mistakenly predict 9 instead of 4, or 1 instead of 7.

Confusion matrices contain sufficient information to calculate a variety of performance metrics, including [**precision**](https://developers.google.com/machine-learning/glossary#precision) and [**recall**](https://developers.google.com/machine-learning/glossary#recall).

# Negative class

In [**binary classification**](https://developers.google.com/machine-learning/glossary#binary_classification), one class is termed positive and the other is termed negative. The positive class is the thing we're looking for and the negative class is the other possibility. For example, the negative class in a medical test might be "not tumor." The negative class in an email classifier might be "not spam." See also [**positive class**](https://developers.google.com/machine-learning/glossary#positive_class).

## Positive class

In [**binary classification**](https://developers.google.com/machine-learning/glossary#binary_classification), the two possible classes are labeled as positive and negative. The positive outcome is the thing we're testing for. (Admittedly, we're simultaneously testing for both outcomes, but play along.) For example, the positive class in a medical test might be "tumor." The positive class in an email classifier might be "spam."

Contrast with [**negative class**](https://developers.google.com/machine-learning/glossary#negative_class).

## True positive (TP)

An example in which the model correctly predicted the [**positive class**](https://developers.google.com/machine-learning/glossary#positive_class). For example, the model inferred that a particular email message was spam, and that email message really was spam.

## True positive rate (TPR)

Synonym for [**recall**](https://developers.google.com/machine-learning/glossary#recall). That is:

True Positive Rate=True Positives/(True Positives+False Negatives)

True positive rate is the y-axis in an [**ROC curve**](https://developers.google.com/machine-learning/glossary#ROC).

## False positive (FP)

An example in which the model mistakenly predicted the [**positive class**](https://developers.google.com/machine-learning/glossary#positive_class). For example, the model inferred that a particular email message was spam (the positive class), but that email message was actually not spam.

## False positive rate (FPR)

The x-axis in an [**ROC curve**](https://developers.google.com/machine-learning/glossary#ROC). The false positive rate is defined as follows:

False Positive Rate=False Positives/False Positives + True Negatives

## True negative (TN)

An example in which the model correctly predicted the [**negative class**](https://developers.google.com/machine-learning/glossary#negative_class). For example, the model inferred that a particular email message was not spam, and that email message really was not spam.

## False negative (FN)

An example in which the model mistakenly predicted the [**negative class**](https://developers.google.com/machine-learning/glossary#negative_class). For example, the model inferred that a particular email message was not spam (the negative class), but that email message actually was spam.

# **Classification: Accuracy**

Accuracy is one metric for evaluating classification models. Informally, **accuracy** is the fraction of predictions our model got right. Formally, accuracy has the following definition:

Accuracy=Number of correct predictions/Total number of predictions

For binary classification, accuracy can also be calculated in terms of positives and negatives as follows:

Accuracy=(TP+TN)/TP+TN+FP+FN

Where *TP* = True Positives, *TN* = True Negatives, *FP* = False Positives, and *FN* = False Negatives.

# **Classification: Precision and Recall**

**Precision** attempts to answer the following question:

What proportion of positive identifications was actually correct?

Precision is defined as follows:

Precision=TP/TP+FP

**Note:** A model that produces no false positives has a precision of 1.0.

|  |  |
| --- | --- |
| True Positives (TPs): 1 | False Positives (FPs): 1 |
| False Negatives (FNs): 8 | True Negatives (TNs): 90 |

Precision=TP/TP+FP=1/1+1=0.5

Our model has a precision of 0.5—in other words, when it predicts a tumor is malignant, it is correct 50% of the time.

## Recall

**Recall** attempts to answer the following question:

What proportion of actual positives was identified correctly?

Mathematically, recall is defined as follows:

Recall=TP/TP+FN

**Note:** A model that produces no false negatives has a recall of 1.0.

|  |  |
| --- | --- |
| True Positives (TPs): 1 | False Positives (FPs): 1 |
| False Negatives (FNs): 8 | True Negatives (TNs): 90 |

Recall=TPTP+FN=11+8=0.11

Our model has a recall of 0.11—in other words, it correctly identifies 11% of all malignant tumors.

# **Classification: ROC Curve and AUC**

ROC curve

An **ROC curve** (**receiver operating characteristic curve**) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:

* True Positive Rate
* False Positive Rate

**True Positive Rate** (**TPR**) is a synonym for recall and is therefore defined as follows:

TPR=TP/TP+FN

**False Positive Rate** (**FPR**) is defined as follows:

FPR=FP/FP+TN

An ROC curve plots TPR vs. FPR at different classification thresholds. Lowering the classification threshold classifies more items as positive, thus increasing both False Positives and True Positives.

**AUC** stands for "Area under the ROC Curve." That is, AUC measures the entire two-dimensional area underneath the entire ROC curve (think integral calculus) from (0,0) to (1,1).

# **Classification: Prediction Bias**

Logistic regression predictions should be unbiased. That is:

"average of predictions" should ≈ "average of observations"

**Prediction bias** is a quantity that measures how far apart those two averages are. That is:

prediction bias=average of predictions−average of labels in data set

**Note:** "Prediction bias" is a different quantity than [bias](https://developers.google.com/machine-learning/crash-course/descending-into-ml) (the b in wx + b).

A significant nonzero prediction bias tells you there is a bug somewhere in your model, as it indicates that the model is wrong about how frequently positive labels occur.

For example, let's say we know that on average, 1% of all emails are spam. If we don't know anything at all about a given email, we should predict that it's 1% likely to be spam. Similarly, a good spam model should predict on average that emails are 1% likely to be spam. (In other words, if we average the predicted likelihoods of each individual email being spam, the result should be 1%.) If instead, the model's average prediction is 20% likelihood of being spam, we can conclude that it exhibits prediction bias.

Possible root causes of prediction bias are:

* Incomplete feature set
* Noisy data set
* Buggy pipeline
* Biased training sample
* Overly strong regularization

You might be tempted to correct prediction bias by post-processing the learned model—that is, by adding a **calibration layer** that adjusts your model's output to reduce the prediction bias. For example, if your model has +3% bias, you could add a calibration layer that lowers the mean prediction by 3%. However, adding a calibration layer is a bad idea for the following reasons:

* You're fixing the symptom rather than the cause.
* You've built a more brittle system that you must now keep up to date.

If possible, avoid calibration layers. Projects that use calibration layers tend to become reliant on them—using calibration layers to fix all their model's sins. Ultimately, maintaining the calibration layers can become a nightmare.

**Note:** A good model will usually have near-zero bias. That said, a low prediction bias does not prove that your model is good. A really terrible model could have a zero prediction bias. For example, a model that just predicts the mean value for all examples would be a bad model, despite having zero bias.

## Bucketing and Prediction Bias

Logistic regression predicts a value between 0 and 1. However, all labeled examples are either exactly 0 (meaning, for example, "not spam") or exactly 1 (meaning, for example, "spam"). Therefore, when examining prediction bias, you cannot accurately determine the prediction bias based on only one example; you must examine the prediction bias on a "bucket" of examples. That is, prediction bias for logistic regression only makes sense when grouping enough examples together to be able to compare a predicted value (for example, 0.392) to observed values (for example, 0.394).

You can form buckets in the following ways:

* Linearly breaking up the target predictions.
* Forming quantiles.

Consider the following calibration plot from a particular model. Each dot represents a bucket of 1,000 values. The axes have the following meanings:

* The x-axis represents the average of values the model predicted for that bucket.
* The y-axis represents the actual average of values in the data set for that bucket.

Rectified Linear Unit (ReLU)

An [**activation function**](https://developers.google.com/machine-learning/glossary#activation_function) with the following rules:

* If input is negative or zero, output is 0.
* If input is positive, output is equal to input.

## activation function

A function (for example, [**ReLU**](https://developers.google.com/machine-learning/glossary#ReLU) or [**sigmoid**](https://developers.google.com/machine-learning/glossary#sigmoid_function)) that takes in the weighted sum of all of the inputs from the previous layer and then generates and passes an output value (typically nonlinear) to the next layer.

## backpropagation

The primary algorithm for performing [**gradient descent**](https://developers.google.com/machine-learning/glossary#gradient_descent) on [**neural networks**](https://developers.google.com/machine-learning/glossary#neural_network). First, the output values of each node are calculated (and cached) in a forward pass. Then, the [**partial derivative**](https://developers.google.com/machine-learning/glossary#partial_derivative) of the error with respect to each parameter is calculated in a backward pass through the graph.

## Gradient descent

A technique to minimize [**loss**](https://developers.google.com/machine-learning/glossary#loss) by computing the gradients of loss with respect to the model's parameters, conditioned on training data. Informally, gradient descent iteratively adjusts parameters, gradually finding the best combination of [**weights**](https://developers.google.com/machine-learning/glossary#weight) and bias to minimize loss.

## multi-class classification

Classification problems that distinguish among more than two classes. For example, there are approximately 128 species of maple trees, so a model that categorized maple tree species would be multi-class. Conversely, a model that divided emails into only two categories (spam and not spam) would be a [**binary classification model**](https://developers.google.com/machine-learning/glossary#binary_classification).

one-vs.-all

Given a classification problem with N possible solutions, a one-vs.-all solution consists of N separate [**binary classifiers**](https://developers.google.com/machine-learning/glossary#binary_classification)—one binary classifier for each possible outcome. For example, given a model that classifies examples as animal, vegetable, or mineral, a one-vs.-all solution would provide the following three separate binary classifiers:

* animal vs. not animal
* vegetable vs. not vegetable
* mineral vs. not mineral

## Softmax Options

Consider the following variants of Softmax:

* **Full Softmax** is the Softmax we've been discussing; that is, Softmax calculates a probability for every possible class.
* **Candidate sampling** means that Softmax calculates a probability for all the positive labels but only for a random sample of negative labels. For example, if we are interested in determining whether an input image is a beagle or a bloodhound, we don't have to provide probabilities for every non-doggy example.

Full Softmax is fairly cheap when the number of classes is small but becomes prohibitively expensive when the number of classes climbs. Candidate sampling can improve efficiency in problems having a large number of classes.

## Softmax

A function that provides probabilities for each possible class in a [**multi-class classification model**](https://developers.google.com/machine-learning/glossary#multi-class). The probabilities add up to exactly 1.0. For example, softmax might determine that the probability of a particular image being a dog at 0.9, a cat at 0.08, and a horse at 0.02. (Also called **full softmax**.)

Contrast with [**candidate sampling**](https://developers.google.com/machine-learning/glossary#candidate_sampling).

